**Exercise 1: Implementing the Singleton Pattern**

public class Logger {

    private static Logger instance;

    private Logger() {

        System.out.println("Logger instance created");

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

        System.out.println("Log: " + message);

    }

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("This is the first log message.");

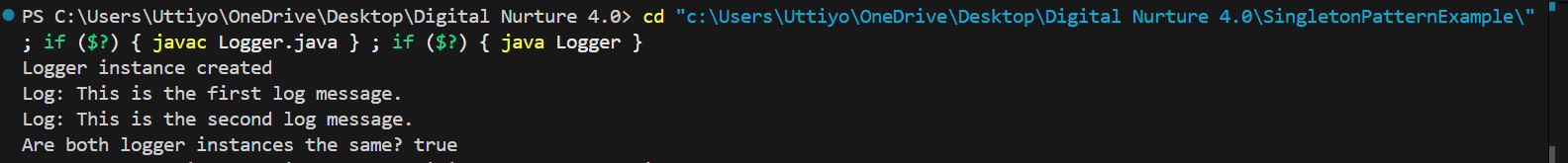
        Logger logger2 = Logger.getInstance();

        logger2.log("This is the second log message.");

        System.out.println("Are both logger instances the same? " + (logger1 == logger2));

    }

}

Output

**Exercise 2: Implementing the Factory Method Pattern**

public class FactoryPatternTest

{

public interface Document {

    void open();

}

static class WordDocument implements Document {

    public void open() {

        System.out.println("Opening Word Document...");

    }

}

static class PdfDocument implements Document {

    public void open() {

        System.out.println("Opening PDF Document...");

    }

}

static class ExcelDocument implements Document {

    public void open() {

        System.out.println("Opening Excel Document...");

    }

}

static class DocumentFactory {

    public Document createDocument(String type)

    {

        if(type.equalsIgnoreCase("Word"))

        {

            return new WordDocument();

        }

        else if(type.equalsIgnoreCase("Pdf"))

        {

            return new PdfDocument();

        }

        else if(type.equalsIgnoreCase("Excel"))

        {

            return new ExcelDocument();

        }

        else

        {

            throw new IllegalArgumentException("Unknown document type: "+type);

        }

    }

}

    public static void main(String[] args) {

        DocumentFactory Factory = new DocumentFactory();

        Document wordDoc = Factory.createDocument("Word");

        wordDoc.open();

        Document pdfDoc = Factory.createDocument("Pdf");

        pdfDoc.open();

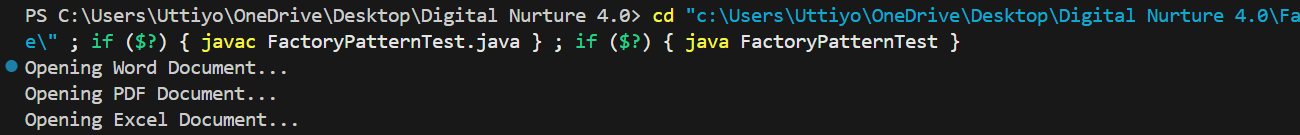
        Document excelDoc = Factory.createDocument("Excel");

        excelDoc.open();

    }

}

Output



**Exercise 3: Implementing the Builder Pattern**

public class Computer {

    private String CPU, RAM, Storage;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.Storage = builder.Storage;

    }

    public static class Builder {

        private String CPU, RAM, Storage;

        public Builder setCPU(String CPU) {

            this.CPU = CPU;

            return this;

        }

        public Builder setRAM(String RAM) {

            this.RAM = RAM;

            return this;

        }

        public Builder setStorage(String Storage) {

            this.Storage = Storage;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    public String toString() {

        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + Storage + "]";

    }

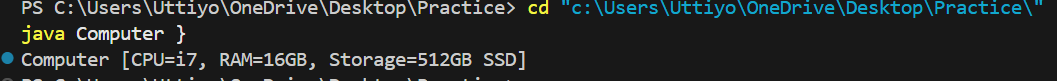
    public static void main(String[] args) {

        Computer comp = new Computer.Builder().setCPU("i7").setRAM("16GB").setStorage("512GB SSD").build();

        System.out.println(comp);

    }

}

Output:

**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

void processPayment();

}

class PayPal {

public void sendPayment() {

System.out.println("Payment processed with PayPal.");

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPal payPal = new PayPal();

public void processPayment() {

payPal.sendPayment();

}

}

public class AdapterDemo {

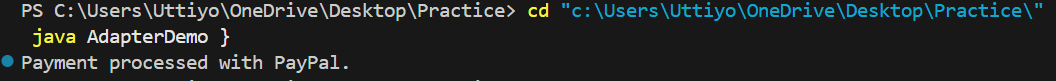
public static void main(String[] args) {

PaymentProcessor processor = new PayPalAdapter();

processor.processPayment();

}

}

**Output:**

**Exercise 5: Implementing the Decorator Pattern**

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

System.out.println("Sending Email Notification");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

}

class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

System.out.println("Sending SMS Notification");

}

}

public class DecoratorDemo {

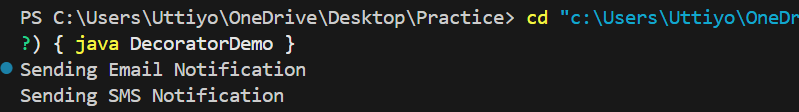
public static void main(String[] args) {

Notifier notifier = new SMSNotifier(new EmailNotifier());

notifier.send();

}

}

**Output:**

**Exercise 6: Implementing the Proxy Pattern**

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + filename);

}

public void display() {

System.out.println("Displaying " + filename);

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

public class ProxyDemo {

public static void main(String[] args) {

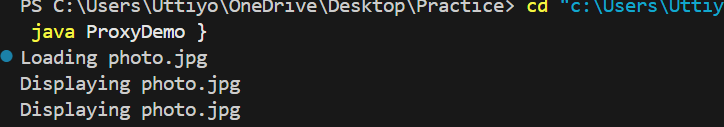
Image image = new ProxyImage("photo.jpg");

image.display(); // loads and displays

image.display(); // only displays

}

}

**Output:**

**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Observer {

void update(String stockName, double price);

}

interface Stock {

void register(Observer o);

void unregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private String stockName;

private double price;

public void setStock(String name, double price) {

this.stockName = name;

this.price = price;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void unregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, price);

}

}

}

class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("MobileApp - " + stockName + " is now $" + price);

}

}

public class ObserverDemo {

public static void main(String[] args) {

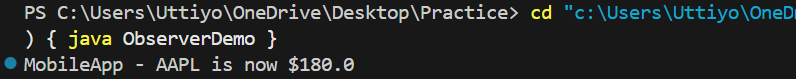
StockMarket stockMarket = new StockMarket();

Observer app = new MobileApp();

stockMarket.register(app);

stockMarket.setStock("AAPL", 180.0);

}

****}  
**Output:**

**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

class PaymentContext {

private PaymentStrategy strategy;

public PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(int amount) {

strategy.pay(amount);

}

}

public class StrategyDemo {

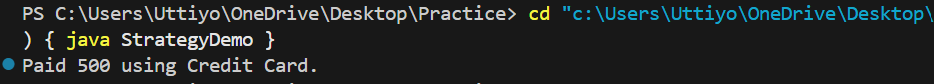
public static void main(String[] args) {

PaymentContext context = new PaymentContext(new CreditCardPayment());

context.executePayment(500);

}

}

**Output:**

**Exercise 9: Implementing the Command Pattern**

interface Command {

void execute();

}

class Light {

void on() { System.out.println("Light is ON"); }

void off() { System.out.println("Light is OFF"); }

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public class CommandDemo {

public static void main(String[] args) {

Light light = new Light();

Command command = new LightOnCommand(light);

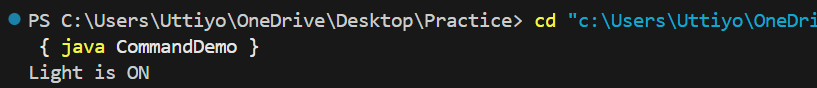
RemoteControl remote = new RemoteControl();

remote.setCommand(command);

remote.pressButton();

}

}

**Output:**

**Exercise 10: Implementing the MVC Pattern**

class Student {

private String name;

private String id;

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getId() { return id; }

public void setId(String id) { this.id = id; }

}

class StudentView {

public void displayStudentDetails(String name, String id) {

System.out.println("Student: " + name + ", ID: " + id);

}

}

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void updateView() {

view.displayStudentDetails(model.getName(), model.getId());

}

}

public class MVCDemo {

public static void main(String[] args) {

Student model = new Student();

model.setName("Alice");

model.setId("S001");

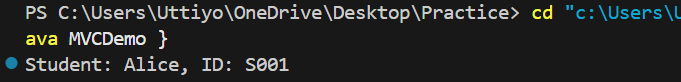
StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

}

}

**Output:**

**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer " + id + " found!";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomer(String id) {

System.out.println(repository.findCustomerById(id));

}

}

public class DIExample {

public static void main(String[] args) {

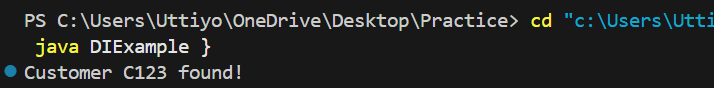
CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomer("C123");

}

}

**Output:**